Algodynamics: algorithms as systems

Venkatesh Choppella*, Kasturi Viswanath† and Mrityunjay Kumar‡
IIIT Hyderabad
India
*venkatesh.choppella@iiit.ac.in, †viswanath.iiithyd@gmail.com, ‡mrityunjay.k@research.iiit.ac.in

***Abstract*—This Full Paper in the Innovative Practice category begins by asking "can algorithms be thought of and taught as dynamical systems?" Our exploration of this idea — Algodynamics — is guided by a vision to achieve convergence between computing and engineering education.**

The engineering sciences share a common conceptual vocabulary originating in dynamical systems: state spaces, flows, actions, invariants, fixed points, convergence, etc. The goal of algodynamics is to build, ab initio, a framework for understanding and teaching algorithms using concepts from dynamics. This allows us to teach computing and algorithms as an engineering science.

Engineers work with models. In algodynamics, models are expressed using transition systems rather than as pseudo-code or programs. This allows a crisp representation of two important classes of computation: sequential algorithms as 'discrete flows' (iterative systems) and interactive applications as 'action flows' (transition systems).

The focus of this paper is on the first of these classes, algorithms, but using ideas from the second, viz., transition systems. In this framework, algorithms emerge as convergent iterative systems. Due to their non-interactivity, iterative systems may be hard to understand. The student may trace through the algorithm, but to know how the parts of the algorithm work together requires 'opening up' the algorithm and situating it within the more general class of interactive systems. Doing so helps the student to understand the machinery of an algorithm in an incremental and modular way. The student interactively solves the algorithmic problem, experimenting with various strategies along the way. At each stage of the design, interactivity is traded for automation.

We illustrate our approach by considering a classic example from sorting: Bubblesort. We first examine a solution based on fixed point iteration. Then we approach Bubblesort as a sequence of five interactive transition systems that culminate in the Bubblesort algorithm. This exercise reveals to the student design decisions and strategies that help understand why Bubblesort works. The successive refinement also pays off in terms of highly modular code whose primitives are elements of the transition system models.

***Index Terms*—algodynamics, algorithms, dynamical systems, transition systems, computing pedagogy, interactive systems.**

I. Introduction

Each domain of engineering relies on a set of scientific principles that are grouped together into *engineering sciences*. For example, mechanical, chemical, and aerospace engineering rely on thermodynamics. Electrical engineering depends on the engineering science of electromagnetics. Aeronautical engineering relies also on the engineering science of aerodynamics.

TABLE I
Engineering Sciences characterised by Engines and Flows.

Engines	Flows	Engineering Science
Heat Engines Locomotives/Automobiles	Heat Flow	Thermodynamics
Electromagnetic Engines Generators/Motors	Current/Magnetic Flow	Electrodynamics
Aero Engines (Aircraft)	Fluid flow	Aerodynamics
Computational Engines (Algorithms/Apps)	Discrete/Action Flow	Algodynamics

The 20th century saw the growth of a new engineering discipline: computing. What is the engineering science underlying computing? An engineering science may be seen as the study of engines, flows, and dynamics. Engines generate or alter flows, and the nature of the flows themselves is governed by dynamical laws, which dictate how something changes over time. Table I is a summary of this view across a few engineering sciences.

The focus of this paper is the last row of Table I, which is about computational engines: algorithms and applications. Algorithms are closed systems that do not interact with their environment. On the other hand, applications are inherently interactive or reactive and exchange information with their environment during their execution. Examples of applications include web and mobile apps, operating systems, routers, protocols, etc. We are interested in the engineering science of computational engines, the flows they generate and the dynamics of these flows. We call the resulting engineering science **Algodynamics**, suggestive of the connection between algorithms and the dynamics inherent in computation.

2021 IEEE Frontiers in Education Conference (FIE) | 978-1-6654-3851-3/21/$31.00 ©2021 IEEE | DOI: 10.1109/FIE49875.2021.9637441

The foundational theories of computer science — Turing Machines and Lambda Calculus — trace their origins to formal logic and the quest to formalise mathematics. However, the levels of abstraction needed to solve most problems of practical interest is rarely matched by the foundational theories. This gap is filled by programming languages. The huge diversity in programming language notations, however, comes in the way of a uniform and portable representation of algorithmic ideas.

Classical models and programming languages built on top of them also face the challenge of how to address interaction, a fundamental paradigm of computing [22]. For the student, however, interaction is a natural part of 'active learning' and 'learning by doing' [40]. Add to that, the student's own experience with computing today (mobile and web apps) is identified more with interaction than with calculation. Interaction, therefore, could be a key route to understanding algorithms, even though the latter are non-interactive.

Algodynamics seeks to be a 'practical theory' for modelling and designing solutions to algorithmic problems. Such a framework should allow the free use of mathematical notation, unencumbered by programming syntax. It should, however, also allow easy translation to programs. It should also incorporate interaction 'ab initio' and treat sequential and interactive systems as part of the same continuum.

Algodynamics is an attempt to bring ideas from theory of dynamical and transition systems, and from what is traditionally called 'formal methods', to help in the presentation and teaching of algorithms. The conceptual vocabulary of dynamics — spaces, flows, actions, fixed points, invariants, etc. — is already part of the engineering sciences. Algodynamics is thus an attempt towards achieving the convergence of computing and engineering education.

The paper advances the goal of Algodynamics by undertaking two tasks. The first task addresses fixed point iteration to solving non-numeric algorithmic problems on data structures. Fixed point iteration is the workhorse of numerical computing [39]. However, it does not merit even a mention in many popular undergraduate texts dealing with algorithms on non-numeric data structures like arrays, trees and graphs [10], [42], [43]. Knuth's "The Art of Computer Programming" Volume 1 [26] defines a 'Computational Method', which is essentially fixed point iteration. It chooses to ignore it, however, in favour of code. On the other hand, the Mapcode methodology [51] applies fixed point iteration on non-numerical algorithms extensively, but seems less well-known.

The goal of the first task is to fill this gap. Our contribution here is twofold: first, a tight definition of what it means for an iterative system to solve an algorithmic problem, and second, a 'design pattern', inspired by dynamical systems and implemented as a higher order function useful for constructing highly modular programs for iterative problem solving.

The second task we undertake is motivated by the observation that interaction is a key step to understanding how something works. Our contribution is here is again twofold: a formal definition of interactive problem solving, along with a systematic methodology for understanding gradual manner with the help of interactive systems based on successive refinement [54] of models, rather than programs. The student is faced with the challenge that algorithms, by their very nature, are non-interactive. Arriving at a deeper understanding requires that the algorithm be 'opened up', its 'moving parts' isolated and individually controlled. The algodynamics approach does this by situating an algorithm in the space of interactive transition systems. Understanding an algorithm requires one to step outside the space of algorithms and into the realm of interactive systems. Through interaction, the student finds herself 'driving' the algorithm to completion rather than merely observing or tracing it. In the process, the student discovers strategies (the sequence of actions) that drive the system to the final (solution) state. Successive refinement of interactive systems incorporates each strategy into the system's dynamics. Interactivity is traded for automation, finally resulting in the algorithm.

This paper is addressed to teachers of computing and curriculum designers. We hope that teachers will benefit from the additional perspective of dynamical systems. Curriculum designers could explore how algodynamics paves a way for a tighter integration between computing and engineering curricula.

This paper grew out of ideas presented in an earlier unpublished technical report [7] and a workshop conducted for college teachers in December 2020 [8].

In the remainder of the paper, we survey related work (Section II), and then present a brief introduction to transition systems (Section III). Iterative and interactive problem solving is presented next (Section IV). Successive refinement of the Bubblesort is used to illustrate the main ideas through an example (Section V). We conclude with a summary of our experience with the Algodynamic approach and with suggestions for future work. Paucity of space limits us to present only the key ideas of algodynamics. Important notions like non-determinism, efficiency, behaviour, system interfaces and system composition are omitted in this paper.

II. Related work: tackling the challenge of learning algorithms

There is a substantial body of literature attesting to the challenges faced by students in problem formulation, solution expression, abstraction, and modelling — key tenets of understanding algorithms and also the challenges in teaching algorithms. Baeza-Yates [4] stresses the role of good toy problems, organizing content based on different paradigms for algorithm design, and use of laboratories to reinforce concepts and encourage experimentation. Using students' mistakes to teach concepts is also another technique that has been found to work [17], [18]. Visual languages [21] and visualizations in general [35], [48] play an important role in teaching algorithms. SOLO (Structure of the Observed Learning Outcome) taxonomy [20] has been suggested to evaluate student's ability to design an algorithm.

Levitin [29] argues for reconsidering the way we teach design and analysis of algorithms and for incorporating more problem-solving where decomposition is a key technique [16]. Another technique is to use the right problems — computational tasks [19] or puzzles [30]. While constructivist approaches have been espoused for teaching programming [5], [55] and constructionism [24] for experiential learning, there is little evidence in the literature of knowledge construction approaches applied to teaching algorithms.

Algorithm visualization systems have been proposed to improve understanding of algorithms [2], [3], [14]. While visualization does offer better engagement and explainability, the outcomes and usage have not been significant enough to label it as an effective approach to teaching algorithms [25], [49], [50]. Naps and coworkers evaluated the reasons for ineffectiveness of these techniques and identified the lack of interaction as the primary reason for ineffectiveness [36]. They proposed a taxonomy to define the level of engagement and interaction. Tracing is another important approach for understanding algorithms. It allows the student to follow, step by step, the execution of an algorithm [31], [32]. While there is evidence that adding interactivity to visualizations can positively impact understanding [37], [52], most of the algorithm visualizations available today lack interactivity, which limits their impact.

The effort closest in spirit to Algodynamics is the Rethinking CS101 project [44], which attempts to build a pedagogy of computing in which interaction is central [45]. The focus of Rethinking CS101 is, however, on programming, not models.

'Learning by doing' is known to build deeper understanding and insight during learning [40]. In the context of algorithms, this means a facility for the student to interact with a system in such a way as to effect a change in its state or the flow of control, along the lines of Category 4 (Changing) and Category 5 (Constructing) of the Engagement Taxonomy introduced by Naps et al. [36]. A key process for scientific knowledge construction is empirical learning, through observation of data and empiricial modelling [13], [41]. Interaction is the key to the student's construction of empirical knowledge. Interacting with the system, the student can empirically discover strategies (i.e., a pattern in the sequence of actions) that lead the system towards the problem's solution.

Meanwhile, leading textbooks on algorithms [10], [42], [43] continue to teach algorithms in a *fait accompli* manner: algorithms are presented in real code or pseudo-code, and the applicable strategy is pointed out ('divide and conquer', 'backtracking', 'greedy', 'dynamic', etc.). Computer Science Curricula 2020 [15] approaches the Draft Competencies in Algorithms and Complexity along similar lines.

Computational Thinking [53] has been advocated as a foundation for computing literacy for school children. Computational thinking has been described as the "thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms" [1]. But how does one express these computational steps? Transition systems provide a well defined and powerful vocabulary and reasoning framework to articulate computational ideas. This is analogous to the vocabulary of *dynamics* (forces, momentum, accelation, mass) useful for reasoning about the mechanical world that students are taught in high school.

III. A brief tour of Transition Systems

The first concern of algodynamics is how to *represent* computations. A good notation for presentation should be compact, unambiguous, portable, amenable to reasoning and easy translation to executable code. In many popular computing text books, computations are expressed either in pseudo-code or in a programming language. The former lacks precision and facility for precise reasoning, as well as standardization [11], while the demands and idiosyncracies of the latter limit the possibility of expressing the model in a compact, abstract and portable way. Therefore, we are encouraged to explore a 'generic' notational and reasoning framework for expressing a computation.

A. Flows and Transition Systems

The simplest way to represent an algorithmic computation is as a discrete-time deterministic dynamical

system, also called a *discrete flow* [51], which is a *state space* X equipped with a *dynamical map* $F : X \to X$. The *trajectory* of a state x is the sequence $x, F(x), F^2(x), \ldots$. x is a *fixed point* of F if $x = F(x)$, it is *transient* otherwise. A solution to a computational problem is obtained by *fixed point iteration*. For each problem instance, a suitable initial state x_0 is located, and a trajectory from x_0 is generated. The problem is solved if the trajectory reaches, in a finite number of iterations, a fixed point of F that holds the answer. Additional machinery on top of the discrete flow is needed to ensure that fixed points exist and are reachable (see Section IV).

The simplest way to describe an interactive system is an *action flow*, which consists of a state space X, a set of actions U and a *transition map* $F : (X, U) \to X$. $x' = F(x, u)$ is written $x \xrightarrow{u} x'$. Interactively solving an algorithmic problem in this setting corresponds to finding a sequence of actions $u_0, \ldots, u_{n-1}$ that steer an initial state x_0 to a final (answer) state x_n via a run $x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} \ldots \xrightarrow{u_{n-1}} x_n$.

The general framework that subsumes both discrete and action flows is that of a (labelled) transition system. A transition system is a six tuple $\langle X, X^0, U, \to, Y, h \rangle$ where X is a set of states with X^0 as a subset of initial states, U is a set of actions, and Y a set of observables. The transition relation $\to$ is a subset of $(X \times U) \times X$. $((x, u), x') \in \to$ is written $x \xrightarrow{u} x'$ and read "x moves to x' on action u", or "x' is a u-successor of x". h is called a *display map*, mapping states to observables. $Post(x, u)$ is the set of all u-successors of x. x is *fixed* if for each u, $Post(x, u) = \{x\}$. $Post(x) = \bigcup_{u \in U} Post(x, u)$. x is *terminal* if $Post(x)$ is empty. $Future(x) = \{(u, x') \mid x \xrightarrow{u} x'\}$ is the set of action-successor pairs (u, x') such that x transits on u to x'.

A system is *deterministic* if $|Post(x, u)| \leq 1$ for each state x and action u. It is *total* if $Post(x, u)$ is nonempty for each x and u. It is *transparent* if $Y = X$ and $h = Id_X$, *agile* if $X = X^0$, *autonomous* if $|U| = 1$ and *interactive* if $|U| \geq 1$. It is *iterative* if it is autonomous, deterministic and total. In an iterative system, the transition relation reduces to a dynamical map $F : X \to X$.

A transition relation is a labelled directed graph whose vertices are states whose labels are drawn from the set U and in which each transition $x \xrightarrow{u} x'$ is an edge from x to x' labelled u. A *run* is a labelled path $\{x_i \xrightarrow{u_i} x_{i+1}\}$ in the graph and its *trajectory* is $\{x_i\}$. If the run is finite, then the last state in the sequence is called the *destination* state. The set of all states in a trajectory is called its *orbit*. An infinite run $\{x_i \xrightarrow{u_i} x_{i+1}\}$ is *convergent* if for some k, $x_i = x_k$ for all $i \geq k$. A run is *initial* if its starting state is initial. A run is *maximal* if it is either infinite or if its destination state is terminal. An initial maximal run is called an *execution*. A system is *terminating* if every execution is finite and *convergent* if every execution is convergent.

Note that an action flow is a transparent, agile, total transition system, and a discrete flow is an action flow that is autonomous.

B. The role of Transition Systems in Computing

Transition systems are to computing what differential equations are to the physical and engineering sciences: a language for expressing change locally and incrementally. Transition systems are versatile tools for modelling almost any phenomenon or process that involves states and actions: interactive or non-interactive, deterministic or non-deterministic, terminating or non-terminating, and sequential or concurrent. In computer science, transition systems are standard tools employed for studying concurrent and hybrid systems [28], [34], [46]. They also appear in various other guises like the operational semantics of programming languages [38]. Transition systems are a way to bridge the apparent dichotomy of 'theory' versus 'systems' that has besetted the discipline of computer science. In the engineering sciences, specially control theory, transition systems over both discrete and continuous domains are staple, and their origin can be traced to systems theory [56].

Despite their ubiquitous presence and long history, transition systems do not seem to be a popular vehicle for expressing sequential algorithms. None of the popular computer science textbooks on algorithms that we sampled [10], [42], [43] refer to transition systems. This is left to the more advanced texts [33], [47]. In short, the potential of transition systems as abstract machines for building mental models of algorithms remains mostly unexplored.

The absence of transition systems in algorithms pedagogy is surprising, given that the connection between algorithms and systems is itself not new at all. As mentioned, it can be traced back at least to the 'computational method' of Knuth [26, p. 7], which was never used in the book and seems to have been largely forgotten since. An adoption and careful treatment of Knuth's method using discrete flows is found in the Mapcode methodology [51]. Dijkstra's formalism of predicate transformers (weakest preconditions) combines non-determinism with guarded commands [12]. The UNITY framework of Chandy and Misra considers a model consisting of a state space along with a finite collection of total transition maps to

support non-determinism [6]. Lamport introduces TLA+, a specification language for writing transition systems [27]. All these methods reduce to the transition systems formalism.

IV. Algorithmic Problem Solving via iterative and interactive systems

This section explores algorithmic problem solving and how it may be approached both in an iterative and an interactive way.

An *algorithmic problem specification* is a tuple $\mathcal{P} = \langle I, A, f, R\rangle$ consisting of an *instance space* I, an *answer space* A, a *functional specification* map $f : I \to A$ and collection R of algebraic operations called *primitives*.

We say a function or relation is *implemented using* R if it is defined as a composition of primitives in R. A transition system is implemented using R if its transition relation and display map are implemented using R. We assume R includes a 'standard' library of datatypes and operations, (e.g., numbers, booleans, finite sequences, arithmetic and relational operators, etc.) and unless necessary, we omit their explicit enumeration.

A. Algorithmic Problem solving using iterative systems

The presentation below synthesizes from the 'Computational Method' [26] and the Mapcode method [51] a design pattern for iterative problem solving expressed as a higher-order combinator *algo_maker*.

Let $\langle X, F\rangle$ be a discrete flow. For each $x \in X$, $orb(x)$ denotes the orbit $\{x, F(x), F^2(x), \ldots\}$. Let $fix(F)$ denote the set of fixed points of F. Let $con(F)$ denote the set of all states that reach a fixed point. Then the *limit map* $F^\infty : con(F) \to fix(F)$ maps each convergent state to its fixed point. The following Python code implements the limit map of a function:

```
def limit_map(F):
    def F_infinity(x): # assume: x in con(F)
        while True:
            xprime = F(x)
            if x == xprime: # fixed point!
                break
            else:
                x = xprime
        return x # guarantee: x in fix(F)
    return F_infinity
```

Given an algorithmic problem $\mathcal{P} = \langle I, A, f, R\rangle$, an *iterative solution* for $\mathcal{P}$ is a tuple $\mathcal{S} = \langle D, \rho, \pi\rangle$ consisting of a discrete flow $D = \langle X, F\rangle$, an initialisation function $\rho : I \to X$ and a projection function $\pi : X \to A$ such that the following conditions are true: 1) **Implementation**: F, ρ and π are are implemented using R. 2) **Convergence**: $\rho(I) \subseteq con(F)$, and 3) **Partial Correctness**: Assuming convergence, $\pi \circ F^\infty \circ \rho = f$.

Informally, to compute $f(v)$ where v is a problem instance, one initialises a system to the initial state $\rho(v)$, then runs the system till it converges to a fixed point. The running of the system is captured via the function F^∞. From the fixed point, the answer is projected using π, so $\pi(F^\infty(\rho(v)))$ is the result of the computation. This should equal $f(v)$, the desired answer. This translates to the following code:

```
def algo_maker(rho,F,pi):
    # F: X->X, rho: I->con(F), pi: fix(F)->A
    F_infty=limit_map(F)
    def f(v):
        return pi(F_infty(rho(v)))
    return f
```

The above formalism captures, in a very general way, a 'design pattern' for fixed point iteration. The pattern is realised via the *limit_map* and *algo_maker* functions.

A function $\beta : X \to W$ for some well-founded relation $(W, \leq)$ is a *bound function* for $F : X \to X$ if $\beta(F(x)) < \beta(x)$ whenever x is transient. A property $P \subseteq X$ is an *invariant* if $x \in P$ implies $F(x) \in P$. If $x^0 \in P$ and x^0 is convergent, then $F^\infty(x^0) \in P$. This relates the initial and the final (fixed) state of the computation. Proving the correctness of iterative problem solving boils down to constructing a suitable bound function and an invariant. The details are omitted. The concepts of bound functions and invariants are the standard machinery of program verification [23], [51].

B. Algorithmic Problem solving using interactive systems

Let $\mathcal{P} = \langle I, A, f, R\rangle$ be an algorithmic problem. Let $v \in I$ be a problem instance. Further, let $S = \langle X, X^0, U, \to, Y, h\rangle$ be an interactive system implemented using R. An *interactive solution* for v using S consists of an execution $x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \ \ldots \xrightarrow{u_{n-1}} x_n$ such that $h(x_n) = f(v)$.

Intuitively, an interactive solution to an instance v of an algorithmic problem $\mathcal{P}$ consists of choosing an initial state x_0 and an execution in S originating in x_0 such that reaches a state x_n whose display is the answer $f(v)$. We may think of this as a single player game where the player constructs an execution $x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \ \ldots \xrightarrow{u_{n-1}} x_n$ with destination x_n whose display $h(x_n)$ is equal to the desired answer $f(v)$. Quite often, the transition system S is deterministic and total, so the transition relation is a transition map $F : X \times U \to X$. In that case, the player needs to only choose the initial state and the action sequence.

The scenario may also be thought of as a laboratory experiment. The experiment is to compute $f(v)$ for different instances v of a problem $\mathcal{P} = \langle I, A, f, R\rangle$. The laboratory has apparatus S built from components in R. The apparatus has a display h and controls U. The

goal of the experiment is to (a) initialise the apparatus and then (b) interact with it through its controls so that it eventually displays $f(v)$.

Let S be a system with state space X and action space U. A *strategy* for S is a function σ that maps each state x of S into a subset of $Future(x)$. The *default strategy* in S is simply $\sigma(x) = Future(x)$. If S is deterministic then only actions are sufficient to specify σ.

A *play* in S with origin x_0 *using* a strategy σ is a run $x_0 \xrightarrow{u_0} x_1 \xrightarrow{u_1} x_2 \ \ldots$ such that for each x_i, $(u_i, x_{i+1}) \in \sigma(x_i)$. The set of all plays from origin x_0 using strategy σ is denoted $\mathcal{Q}_\sigma(x_0)$.

Let $\mathcal{P} = \langle I, A, f, R \rangle$ be a problem and S a transition system whose transition function is built using the primitives in R . An *interaction scheme* in S for $\mathcal{P}$ is a pair $\langle \hat{\rho}, \sigma \rangle$ where $\hat{\rho} : I \to 2^{X^0}$ is an initialisation relation and σ is a strategy for S. $\langle \hat{\rho}, \sigma \rangle$ is a

1) *feasible solution scheme* for $\mathcal{P}$ with S if for each $v \in I$, there exists an $x_0 \in \hat{\rho}(v)$ and a play p in $\mathcal{Q}_\sigma(x_0)$ such that p visits a state x_n such that $h(x_n) = f(v)$.
2) *guaranteed solution scheme* for $\mathcal{P}$ with S if for each $v \in I$, for each $x_0 \in \hat{\rho}(v)$ and for each play p in $\mathcal{Q}_\sigma(x_0)$, p visits a state x_n such that $h(x_n) = f(v)$.

V. Example: Bubblesort

We apply Algodynamics to the Bubblesort sorting algorithm. We choose Bubblesort not due its efficiency, which is poor, but because it is small enough, yet non-trivial, and lends itself to a detailed analysis of its dynamics.

We consider arrays as finite sequences of natural numbers and denote the set of arrays $\mathbb{N}^*$. An array of size n is a map from $[0, \ldots, n-1]$ to $\mathbb{N}$. $sort(a)$ denotes the sorted permutation of array a and $sorted?(a)$ determines if a is sorted. Let R consist of the following operations: arithmetic over naturals, comparison over elements of naturals, array reference and assignments. The sorting problem may be specified as $\mathcal{P} = \langle I, A, f, R \rangle$ where $I = A = \mathbb{N}^*$, $f = sort$, R as specified above.

A. Iterative solution

The solution for $\mathcal{P}$ is $\langle D, \rho, \pi \rangle$ in which $D = \langle X, F : X \to X \rangle$, $X = \mathbb{N} \times N \times \mathbb{N}^*$ with the state vector $x \in X$ written as a triple (i, b, a). $\rho(a^0) = (0, |a^0|, a^0)$ and $\pi(i, b, a) = a$. F is given below:

```
def swap(a,i,j):
    (a[i], a[j]) = (a[j], a[i])
    return a

def F(x):
    [a, i, b] = x
    if (b == 0):
        return x
    elif (i == b-1):
        return [a, 0, b-1]
    elif (a[i] <= a[i+1]):
        return [a, i+1, b]
    else: # a[i] > a[i+1]:
        return [swap(a,i,i+1), i+1, b]
```

The solution is complete once we define the function ρ and π and put everything together:

```
def rho(a):
    return [a, 0, len(a)]

def pi(x):
    [a, i, b] = x
    return a

def sort(a):
    return algo_maker(rho, F, pi)(a)
```

Here is an example of sorting [8 6 7 4]. To save space, the index i is indicated by the position of the underlined element, and b by the position of the bar.

$$[\underline{8}\ 6\ 7\ 4 \mid] \to [6\ \underline{8}\ 7\ 4 \mid] \to [6\ 7\ \underline{8}\ 4 \mid] \to [6\ 7\ 4\ \underline{8} \mid] \to$$
$$[\underline{6}\ 7\ 4 \mid 8\ ] \to [6\ \underline{7}\ 4 \mid 8\ ] \to [6\ 4\ \underline{7} \mid 8\ ] \to [\underline{6}\ 4 \mid 7\ 8\ ] \to$$
$$[4\ \underline{6} \mid 7\ 8\ ] \to [\underline{4} \mid 6\ 7\ 8\ ] \to [\mid \underline{4}\ 6\ 7\ 8\ ]$$

Bound function: The function $B(i, b, a) \stackrel{\text{def}}{=} (b, b - i)$ is a bound function for $orb(0, n, a^0)$, where $n = |a^0|$. (The proof is omitted.)

Invariant: Consider the property $P \subseteq X$ defined as the set of all $x = (i, b, a)$ such that:

1) $0 \leq b \leq n$ where $n = |a|$
2) $0 \leq i < b$
3) $a_i = max\{a_j \mid 0 \leq j \leq i\}$
4) $sorted?\ a[b..n-1]$
5) $\forall k : b \leq k < n, \forall j : 0 \leq j < b, a_j \leq a_k$

Verifying that P is invariant and that the initial state $(0, |a^0|, a)$ satisfies P is omitted.

B. Interactive solution and successive refinement

We build a gradual understanding of Bubblesort by opening it up into a series of five deterministic interactive transition systems 'machines' B_1 to B_5 connected via successive refinement [9] with B_5 corresponding to the Bubblesort algorithm. The design path is not fixed for the algorithm, rather, it is a function of the granularity of algorithm's state space, strategies involved, choice of pedagogy, etc.

In what follows, a^0 is the problem instance, n is $|a^0|$ and a is a state variable ranging over arrays, initialised to a^0. i and j are array indices, with $0 \leq i < j < n$ and $0 \leq b \leq n$. i and b appear as part of the state vector in some of the transition systems $B_1 - B_5$. For all the systems, the observables are arrays and the display

h projects the array from the state. Since $B_1 - B_5$ are deterministic, strategies are written as sets of actions. Each machine is described below:

1) B_1: "Swap Machine": The core of Bubblesort is swapping elements in an array. B_1 is designed around this observation. The state vector of B_1 consists of the array variable a. An action in B_1 is of the form $\mathsf{swap}(i,j)$, where $0 \leq i < j < n$. The transition relation of B_1 is $a \xrightarrow[B_1]{\mathsf{swap}(i,j)} a'$ iff $a' = swap(a,i,j)$.

Here are example runs in B_1 starting with the array $[8,6,7,4]$. The underlines identify elements being swapped.

$$[\underline{8},6,7,\underline{4}] \xrightarrow[B_1]{\mathsf{swap}(0,3)} [4,6,7,8] \quad (1)$$

$$[8,\underline{6},\underline{7},4] \xrightarrow[B_1]{\mathsf{swap}(1,2)} [\underline{8},7,6,\underline{4}] \xrightarrow[B_1]{\mathsf{swap}(0,3)} [4,7,6,8] \quad (2)$$

Note that B_1 is interactive, deterministic and non-terminating. We are free to swap any two elements, and this is the default strategy. This can sometimes result in very short solution runs, as in (1). The freedom also makes B_1 versatile. B_1 can not just sort, but also reverse a sequence, as in (2). With some practice, the student could conjecture that B_1 may be used to obtain *any* permutation of the array. The connection with the theory of permutation groups may be explored. The student may conjecture and verify that interaction scheme with $\hat{\rho}_{B_1}(a) = \{a\}$ and the default strategy is a feasible but not guaranteed solution scheme for the sorting problem using B_1.

B_1 is not a guaranteed solution because it allows too much freedom and lacks direction. Playing with B_1, the student might realise that sorting can be done by swapping only those elements that are out of order ('ordering'). She may construct the strategy σ_{B_1} where $\sigma_{B_1}(a) = \{\mathsf{swap}(i,j) \mid i < j \text{ and } a_i > a_j\}$. and then show that $\langle \hat{\rho}_{B_1}, \sigma_{B_1} \rangle$ is a guaranteed solution for sorting.

2) B_2: "Order Machine": The system B_2 incorporates the strategy σ_{B_1} into its dynamics. B_2 has the array a as its sole state variable. The sole action of B_2 is $\mathsf{order}(i,j)$, $0 \leq i < j < n$. The transition relation is $a \xrightarrow[B_2]{\mathsf{order}(i,j)} a'$ iff $i < j$ and $a_i > a_j$ and $a' = swap(a,i,j)$.

Interacting with the virtual experiment implementing B_2 should help the student conjecture and then prove that B_2's default strategy along with the initialisation $\hat{\rho}(a) = \{a\}$ is a guaranteed solution, i.e, that every execution in B_2 reaches a sorted array (and also terminates there).

Is it now possible to have a strategy for picking the indices for ordering? The simplest strategy picks adjacent elements. This is captured by the strategy $\sigma_{B_2}(a) = \{\mathsf{order}(i,j) \in Future(a) \mid j = i+1\}$.

This strategy along with the identity initialisation is also a guaranteed solution for sorting a^0. The proof is a simple exercise for the student.

3) B_3: "Order Adjacent machine": The transition system B_3 incorporates the strategy σ_{B_2}. B_3 has as its state variable an array a and a single action $\mathsf{adj}(i)$ where $0 \leq i < n-1$. Its dynamics is described by the transitions in B_2: $a \xrightarrow[B_3]{\mathsf{adj}(i)} a'$ iff $a \xrightarrow[B_2]{\mathsf{order}(i,i+1)} a'$.

B_3 still leaves undecided which action $adj(i)$ is picked from the available choices. A simple strategy is to choose the action $adj(i)$ with the smallest i amongst all the available actions. Another strategy is to sweep the array left to right, keeping track of an index i, $0 \leq i < n-1$, examining the pair a_i and a_{i+1}, and swapping them if necessary, and in either case, advancing i. The student will soon realise that *multiple sweeps* of the array are necessary, making it necessary to be able to 'reset' i to zero.

To implement this strategy, an additional state variable i and action reset will be needed. This takes us to the system B_4.

4) B_4: "Bubble" Machine: The system B_4 has at its state vector the pair (a,i) with array a and index variable i, $0 \leq i < n-1$, where $n = |a|$. The initial state of B_4 is $(a,0)$. The action set of B_4 consists of inc and reset. inc orders the elements at i and $i+1$ and increments i. The reset resets i to zero, heralding the beginning of a new sweep cycle. The transition relation of B_4 is $(a,i) \xrightarrow[B_4]{\mathsf{reset}} (a,0)$, $(a,i) \xrightarrow[B_4]{\mathsf{inc}} (a,i+1)$ iff $i < n-1$ and $a_i \leq a_{i+1}$, and $(a,i) \xrightarrow[B_4]{\mathsf{inc}} (a',i+1)$ iff $i < n-1$ and $a \xrightarrow[B_3]{\mathsf{adj}(i)} a'$.

Playing with B_4 by tracing out a few runs, an observant student may notice the following phenomenon: the index i carries along with it the maximum element in the array swept so far. The largest element 'bubbles' its way towards the right end of the array. The bubbling may be interrupted by a reset at any step. Furthermore, the reset could be invoked infinitely often. B_4 is thus no longer a terminating system. Playing some more with the experiment, a student could then make the crucial observation that repeated sweeps of the B_4 machine can be used to arrange the maximum element of an array at position $n-1$, the next maximum at position $n-2$, and so on. This divides the array at any step into an unsorted part followed by a sorted part. Whenever the sweep index i reaches the boundary between the two parts, a reset could be triggered. However, to implement this strategy, an additional index would be needed as part of the transition that determines the boundary between the unsorted and sorted. This motivates the definition of the next (and final) transition system B_5.

5) B_5: "Bubblesort" machine: B_5 incorporates into its dynamics the strategy for resetting i. The state vector of B_5 now includes an additional *boundary* variable b, which ranges from 0 to n. A boundary value of b means that all elements from a_b to a_{n-1} are sorted and are greater than or equal to all elements a_0 to a_{b-1}. The state vector (a, i, b) is initialised to $(a^0, 0, n)$. The sweep index i varies from 0 to $b-1$. At each step, the sweep index i is incremented, until it reaches $b-1$, at which time it is reset to 0. Simultaneously, the boundary index b is decremented. The system terminates when b is 0. B_5 is an iterative system with one action next. The dynamics of B_5 is defined below. It is worth comparing it with the transition function F in (Section V-A).

$$(a,i,b) \xrightarrow[B_5]{\mathsf{next}} (a',i',b) \quad \text{iff } b > 0 \wedge$$
$$i < b-1 \ \wedge (a,i) \xrightarrow[B_4]{\mathsf{inc}} (a',i')$$
$$(a,i,b) \xrightarrow[B_5]{\mathsf{next}} (a',i',b-1) \quad \text{iff } b > 0 \wedge$$
$$i = b-1 \wedge \ (a,i) \xrightarrow[B_4]{\mathsf{reset}} (a',i')$$

Whenever i reaches $b-1$, it is automatically reset, b is decremented and as a result, the sorted segmented of the array grows by one. Formal correctness proofs are omitted. B_5 terminates just when the iterative system based on the function F reaches a fixed point.

C. Coding the transition systems into a program

In the implementation below, actions translate to functions of state variables. Notice the absence of nested while loops. A comparison with the iterative solution is instructive.

```
# action from B_1
def swap(a, i, j):
    a[i], a[j] = a[j], a[i]
    return a

# action from B_2
# assumes a[i]>a[j]
def order(a, i, j):
    return swap(a, i, j)

# action from B_3
# assumes a[i]>a[i+1]
def adj(a, i):
    return order(a, i, i+1)

# action from B_4
def inc(a, i):
    if (a[i] > a[i+1]):
        return (adj(a, i), i+1)
    else:
        return (a, i+1)

# action from B_4
def reset(a, i):
    return (a, 0)

# action from B_5
def next(a, i, b):
    if (i < b-1):
        (a, i) = inc(a, i)
        return (a, i, b)
    else:
        (a, i) = reset(a, i)
        return (a, i, b-1)

# Bubblesort
def bubblesort(a):
    (i, b)    = (0, len(a))
    while (b > 1):
      (a, i, b) =  next(a, i, b)
    return a
```

VI. Future work and Conclusions

Algodynamics situates algorithms within dynamical systems. We have briefly explored its theory, design patterns and iterative and interactive problem solving methodologies in this paper, leaving many more aspects, like complexity, concurrency and distribution for future work.

We have studied a variety of data structures with algodynamics, and explored several problem solving paradigms: divide and conquer, recursion, dynamic programming, and the greedy strategy and have developed over 20 experiment prototypes. These results will be reported elsewhere.

Algodynamics allows thinking about problem solving both iteratively and interactively. It encourages teachers to construct their own pathways of transition systems to 'open up' their favourite algorithms.

Algodynamics could be the engineering science of computing from which several of its sub-disciplines might benefit. Exploring this possibility is a rich opportunity for teachers and curriculum designers.

References

[1] A V Aho. Computation and computational thinking. *Ubuity*, 2011.

[2] Ali Alharbi, Frans Henskens, and Michael Hannaford. Integrated Standard Environment for the Teaching and Learning of Operating Systems Algorithms Using Visualizations. In *2010 Fifth International Multi-conference on Computing in the Global Information Technology*, pages 205–208, Valencia, Spain, September 2010. IEEE.

[3] Saleema Amershi, Giuseppe Carenini, Cristina Conati, Alan K. Mackworth, and David Poole. Pedagogy and usability in interactive algorithm visualizations: Designing and evaluating CIspace. *Interacting with Computers*, 20(1):64–96, January 2008.

[4] Ricardo A Baeza-Yates. Teaching algorithms. *ACM SIGACT News*, 26(4):51–59, 1995.

[5] Mordechai Ben-Ari. Constructivism in computer science education. *Journal of Computers in Mathematics and Science Teaching*, 20(1):45–73, 2001.

[6] K M Chandy and J Misra. *Parallel Program Design*. Addison-Wesley, 1988.

[7] Venkatesh Choppella, Viswanath Kasturi, Mrityunjay Kumar, and Ojas Mohril. Algodynamics: Teaching algorithms using interactive transition systems. https://arxiv.org/abs/2010.10015, 2020.

[8] Venkatesh Choppella, Viswanath Kasturi, Mrityunjay Kumar, and Ojas Mohril. Algodynamics: Teaching algorithms using interactive transition systems, 2020. Workshop presented at ACM Compute 2020.

[9] Venkatesh Choppella, Ojas Mohril, and Archit Goel. Online demo of bubblesort. https://algodynamics.io/bubblesort, 2021.

[10] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[11] Quintin Cutts, Richard Connor, Greg Michaelson, and Peter Donaldson. Code or (not code): Separating formal and natural language in cs education. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, WiPSCE 14, pages 20–28. ACM, 2014.

[12] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[13] Rosalind Driver, Hilary Asoko, John Leach, Philip Scott, and Eduardo Mortimer. Constructing scientific knowledge in the classroom. *Educational researcher*, 23(7):5–12, 1994.

[14] Clifforda Shaffer et al. Algorithm Visualization: The State of the Field. *ACM Transactions on Computing Education*, 10(3):1–22, August 2010.

[15] Association for Computing Machinery (ACM) and IEEE Computer Society (IEEE-CS). *Computing Curricula 2020: Paradigms for Global Computing Education*. ACM, 2020. https://www.acm.org/binaries/content/assets/education/curricula-recommendations/cc2020.pdf.

[16] David Ginat. On varying perspectives of problem decomposition. *ACM SIGCSE Bulletin*, 34(1):331–335, 2002.

[17] David Ginat. The greedy trap and learning from mistakes. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 11–15, 2003.

[18] David Ginat. Learning from wrong and creative algorithm design. *ACM SIGCSE Bulletin*, 40(1):26–30, 2008.

[19] David Ginat, Dan Garcia, Owen Astrachan, and Joseph Bergin. Colorful illustrations of algorithmic design techniques and problem solving. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 425–426, 2001.

[20] David Ginat and Eti Menashe. Solo taxonomy for assessing novices' algorithmic design. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, page 452457, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Daniela Giordano and Francesco Maiorana. Teaching algorithms: Visual language vs flowchart vs textual language. In *2015 IEEE Global Engineering Education Conference (EDUCON)*, pages 499–504. IEEE, 2015.

[22] Dina Goldin, Scott A. Smolka, and Peter Wegner, editors. *Interactive Computation: The New Paradigm*. Springer–Verlag, 2006.

[23] David Gries. *The Science of Programming*. Springer, 1981.

[24] Idit Ed Harel and Seymour Ed Papert. *Constructionism*. Ablex Publishing, 1991.

[25] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13(3):259–290, June 2002.

[26] Donald Knuth. *The art of computer programming (taocp)*. Addison-Wesley, 2011. First published 1968.

[27] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, 2002.

[28] Edward A. Lee and Sanjit A. Seshia. *Introduction to Embedded Systems*. http://LeeSeshia.org, 2nd edition, 2015.

[29] Anany Levitin. Design and analysis of algorithms reconsidered. *ACM SIGCSE Bulletin*, 32(1):16–20, 2000.

[30] Anany Levitin and Mary-Angela Papalaskari. Using puzzles in teaching algorithms. In *Proceedings of the 33rd SIGCSE technical symposium on computer science education*, pages 292–296, 2002.

[31] Raymond Lister, Colin Fidge, and Donna Teague. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *ACM SIGCSE Bulletin*, 41(3):161–165, 2009.

[32] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. Relationships between reading, tracing and writing skills in introductory programming. In *Proceeding of the fourth international workshop on Computing education research - ICER '08*, pages 101–112, Sydney, Australia, 2008. ACM Press.

[33] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[34] Robin Milner. *A Calculus of Communicating Systems*. Springer Verlag, 1980.

[35] Thomas Naps et al. Exploring the role of visualization and engagement in computer science education. In *Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 131–152. 2002.

[36] Thomas Naps et al. Evaluating the educational impact of visualization. In *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ITiCSE-WGR 03, pages 124–136. ACM, 2003.

[37] Mrinal Patwardhan and Sahana Murthy. When does higher degree of interaction lead to higher learning in visualizations? Exploring the role of Interactivity Enriching Features. *Computers & Education*, 82:292–305, March 2015.

[38] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[39] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.

[40] Maria Roussou. Learning by doing and learning through play: An exploration of interactivity in virtual environments for children. *Comput. Entertain.*, 2(1):10, January 2004.

[41] Steve Russ. Empirical modelling: The computer as a modelling medium. *Computer Bulletin*, 39(2):20–22, 1997.

[42] Sartaj Sahni and Ellis Horowitz. *Fundamentals of computer algorithms*. Computer Science Press, 1978.

[43] Robert Sedgewick. *Algorithms*. Pearson Education, 1988.

[44] L. A. Stein. The Rethinking CS 101 Project. http://cs101.org, 2005. (last accessed: 2020-01-21).

[45] L. A. Stein. Interaction, computation and education. pages 463–484. 2006. in [22].

[46] Paulo Tabuada. *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer, 2009.

[47] Gerard Tel. *Introduction to Distributed Algorithms. 2nd Ed.* Cambridge University Press, 2000.

[48] Jaime Urquiza-Fuentes and J Ángel Velázquez-Iturbide. A survey of successful evaluations of program visualization and algorithm animation systems. *ACM Transactions on Computing Education (TOCE)*, 9(2):1–21, 2009.

[49] Jaime Urquiza-Fuentes and J. Ángel Velázquez-Iturbide. A Survey of Successful Evaluations of Program Visualization and Algorithm Animation Systems. *ACM Transactions on Computing Education*, 9(2):1–21, June 2009.

[50] Ladislav Végh and Veronika Stoffová. Algorithm Animations for Teaching and Learning the Main Ideas of Basic Sortings. *Informatics in Education*, 16(1):121–140, March 2017.

[51] Kasturi Viswanath. *An introduction to Mathematical Computer Science*. The Universities Press, 2008.

[52] Pei-Yu Wang, Brandon K. Vaughn, and Min Liu. The impact of animation interactivity on novices learning of introductory statistics. *Computers & Education*, 56(1):300–311, January 2011.

[53] Jeannette M Wing. Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3717–3725, 2008.

[54] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[55] Tom Wulf. Constructivist approaches for teaching computer programming. In *Proceedings of the 6th Conference on Information Technology Education*, SIGITE 05, page 245248, New York, NY, USA, 2005. Association for Computing Machinery.

[56] L. A. Zadeh. From circuit theory to system theory. *Proceedings of the IRE*, 50(5):856–865, 1962.